

---

# In Situ Analysis and Visualization with Catalyst and Ascent

Jean M. Favre,  
Senior Visualization Software Engineer

May 16, 2023

# Thanks to different teams

- Thanks to the many actors in the visualization and in-situ dev teams:
- Berkeley Lab, ANL, LLNL, Los Alamos, Sandia, and many others (in particular Cyrus Harrison)

# Morning agenda

- 09:30      Welcome, Overview and Motivation  
            Agenda and technical details for demonstrations
- 09:45      Introduction to in-situ visualization, workflows and terminology
- 10:15      Conduit, an API to describe hierarchical scientific data  
            The Mesh Blueprint, usage conventions, examples
- 10:45      Coffee break
- 11:00      Ascent, an in-situ visualization and analysis library using Conduit
- Making images
  - Transforming data, extracting data
  - Queries and Conditional triggers
- 11:30      Instrumentation examples with Ascent
- Jacobi{C++/Python}
  - SPH-EXA: an open-source parallel particle simulation code
  - LULESH: an open-source parallel unstructured flow solver

# Afternoon agenda

13:00          ParaView Catalyst v2, an in-situ visualization and analysis library using Conduit

- The ParaView interactive application
- The Paraview parallel server architecture
- The Catalyst API, and the ParaView Catalyst Blueprint
- Python scripting, Data Extractors

14:00          Instrumentation examples with Catalyst

- Jacobi{C++/Python}
- SPH-EXA: an open-source parallel particle simulation code
- LULESH: an open-source parallel unstructured flow solver
- Catalyst, connecting to a live simulation, steering

15:00          Coffee break

15:30          Ascent executing ParaView Python code

16:35          Future developments, alternative workflows

- ADIOS2 + Fides + ParaView

16:45-17:00   Wrap-up, Q&A

## Technical details

- course account “class5?” with a password
- `ssh ela.cscs.ch + ssh daint.cscs.ch`
- <https://jupyter.cscs.ch>
- Your account has been already bootstrapped with requirements for jupyter lab
  - see `$HOME/jupyterlab-cscs.env`
- Reservation for computer nodes “insitu”
- Your account will expire at the end of the day
- My repository should not expire: <https://github.com/jfavre/InSitu-Vis-Tutorial2022>

`cd InSitu-Vis-Tutorial2022; git pull`

N.B. The commands in “PizDaint\_Instructions.2023.txt” should already have been done for your account.

# Overview

- What is in-situ visualization, why do we need it? What solutions are available to implement it?
- See recent book “[In Situ Visualization for Computational Science](#)”
- Some examples from the previous decade (libSim, Catalyst)
- Replay some of the presentations from the most recent Ascent tutorial
- Detail the ParaView-Catalyst and Ascent solutions, with practical examples.

# “Piz Daint” at CSCS, the Swiss flagship for national HPC Service

- Cray XC40/XC50
- 5704 hybrid nodes (Intel Xeon E5-2690 v3/NVIDIA Tesla P100)
- 1813 multi-core nodes (Intel Xeon E5-2695 v4)



Coming online in 2023, the ‘Alps’ system infrastructure will replace CSCS’s Piz Daint and serve as a general-purpose system.

[Link](#)





# The past

- For decades, the dominant paradigm has been *post-hoc* visualization
- Simulation codes iterate, and save data at regular time intervals.
  - Visualization and domain scientists can then read the data back from storage and interactively explore the data without time constraints
  - In particular at CSCS, see <https://user.cscs.ch/computing/visualisation/>

“Without I/O, no visualization is possible”

The true cost of doing I/O is an aggregate of the solver’s I/O phase and the many iterations of visualization sessions.

# Post-hoc visualization

Even if scientists could afford to keep most of the data for analysis, they must transfer the data to a machine with sufficient capacity and processing power:

- Very high data transfer
  - ➔ Visualization machine needs to be almost as powerful as the supercomputer
- The alternative: use smaller temporal and spatial subsets

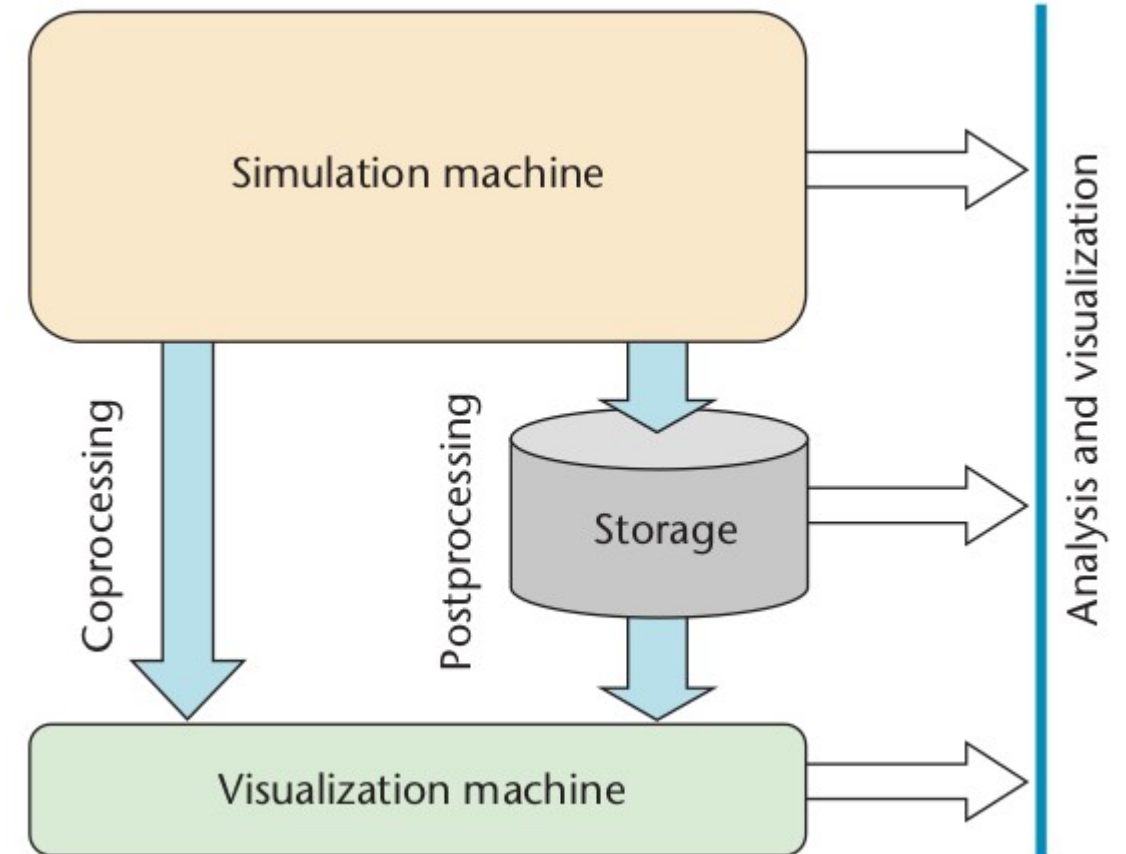


Figure taken from “*In Situ Visualization at Extreme Scale: Challenges and Opportunities*”, Kwan-Liu Ma, IEEE CG&A, nov/dec 2009

# in situ visualization

Instrument the code such that both the simulation and visualization calculations run on the same hardware

This runtime co-processing can render images directly or extract features -- *which are much smaller than the original raw data*

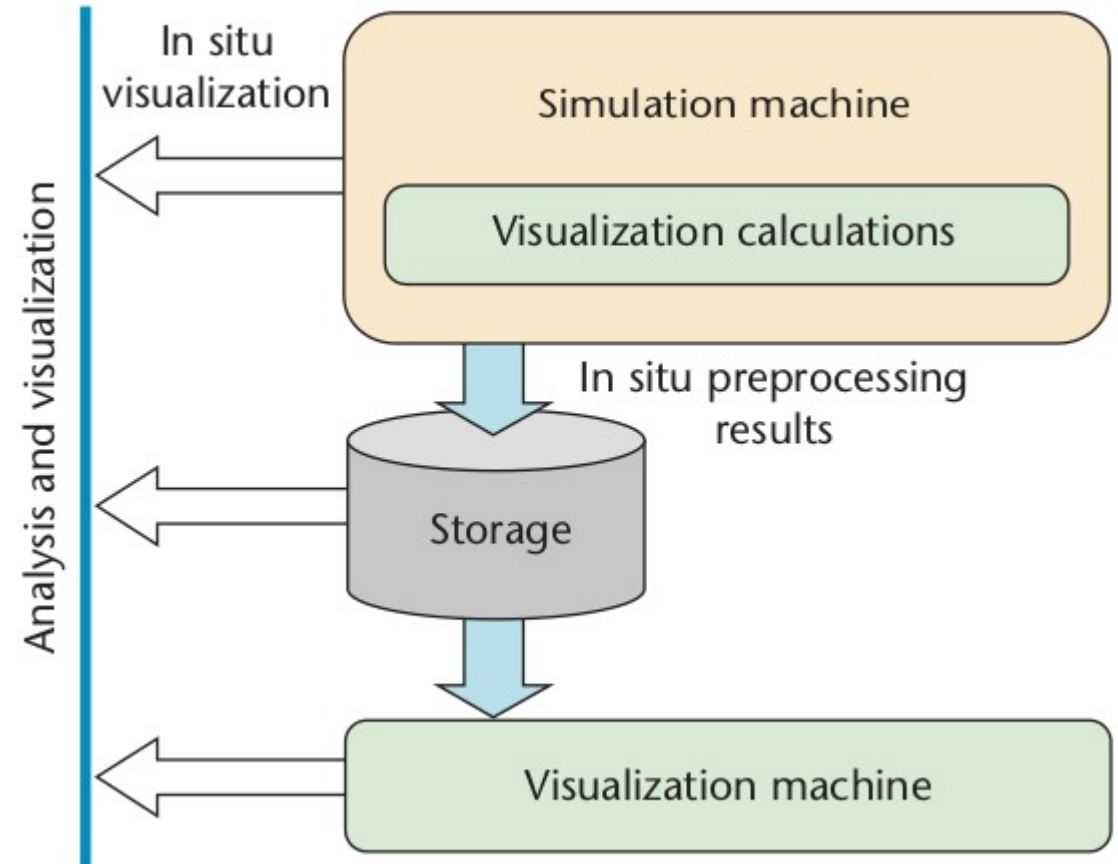


Figure taken from "In Situ Visualization at Extreme Scale: Challenges and Opportunities", Kwan-Liu Ma, IEEE CG&A, nov/dec 2009

# In-situ visualization has raised quite a few questions

- Sharing physical resources and domain decomposition?
- What % of time can we afford to “do visualization” vs. “advance the solver”?
- Which feature extraction and visualization tasks are best suited for on-the-fly processing?
- Since less data would be effectively stored to disk, should we augment it with ancillary data?
- Can we provide a generic abstraction to describe the data and mesh structures?

# A third paradigm also emerged: in-transit visualization

## Processing paradigms for scientific visualization

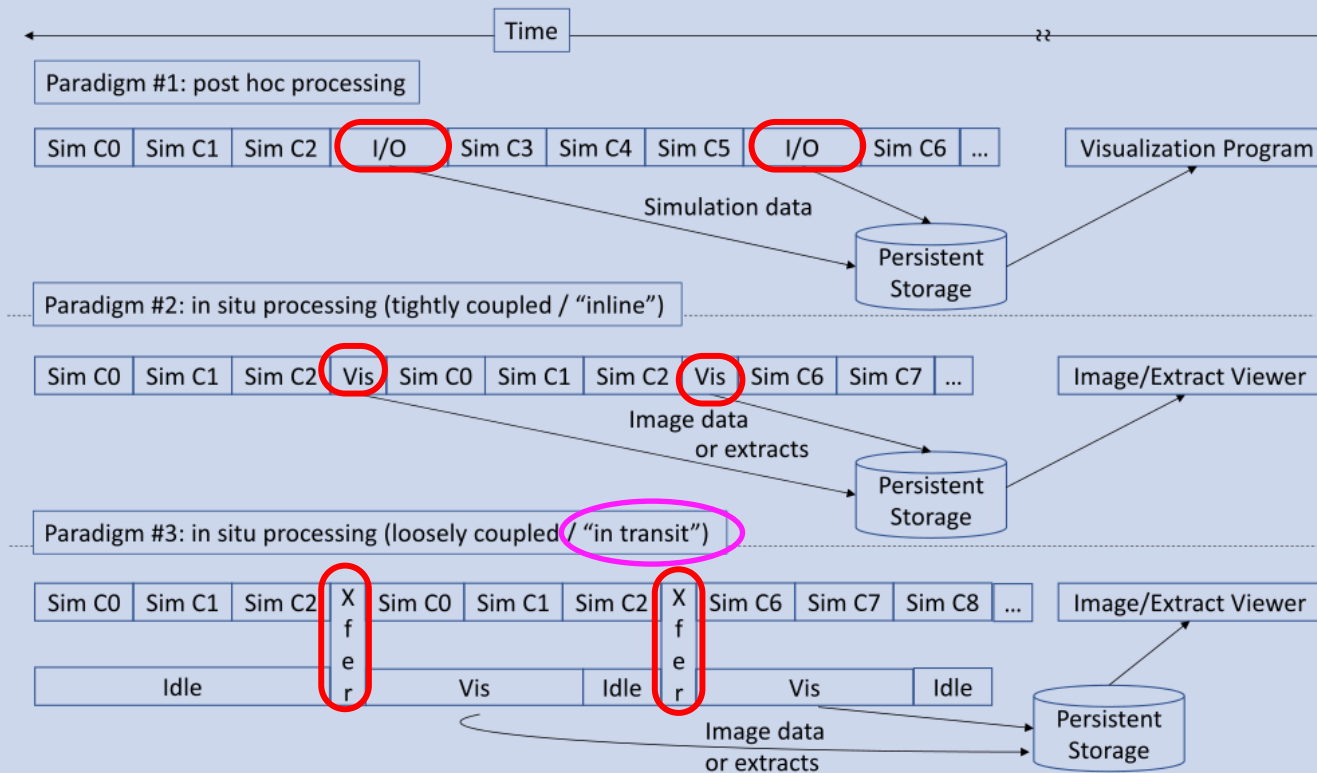


Figure taken from "In Situ Visualization for Computational Science", Hank Childs et al., IEEE CG&A, nov/dec 2019

# Does “in-situ” mean “in place”?

- The data is already in the processor’s memory space, without touching the disks
- If simulation data is moved to a distinct set of resources (nodes dedicated to visualization), we are still analysing data “in place”, but is it “in-situ”?
- There has been quite a few variants on the terminology:
  - Co-processing, concurrent processing, run-time visualization

## Many definitions and colloquial use for “in-situ”

- An exhaustive panorama of the different systems in use was created :
- "A Terminology for In Situ Visualization and Analysis Systems“, Hank Childs et al, International Journal of High Performance Computing Applications, 34(6):676–691
- <http://cdux.cs.uoregon.edu/pubs/ChildsIJHPCA.pdf>
- For the scope of this paper, “in situ processing” was defined to be:

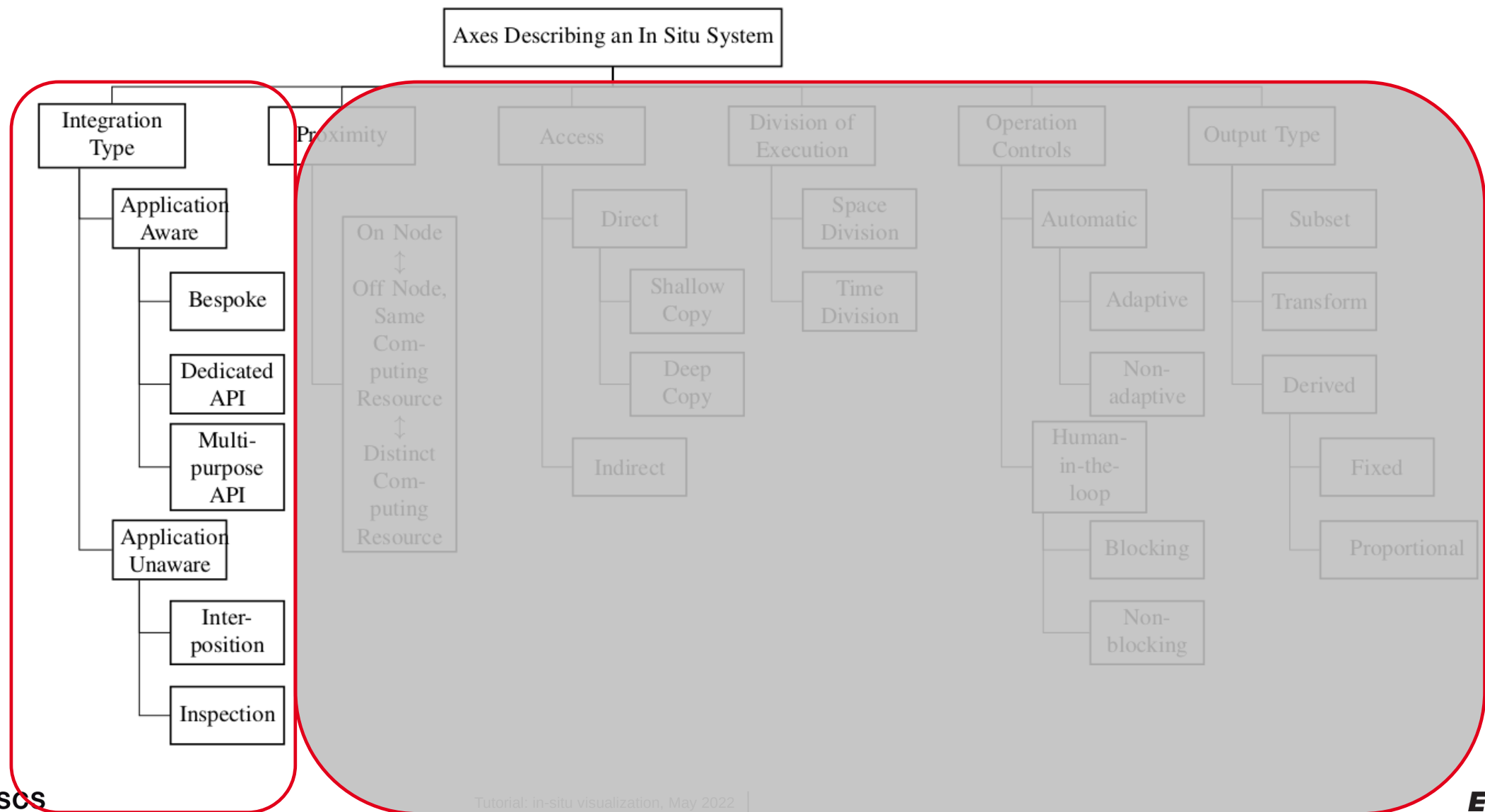
“processing data as it is generated”

# in situ systems were best described via multiple, distinct axes

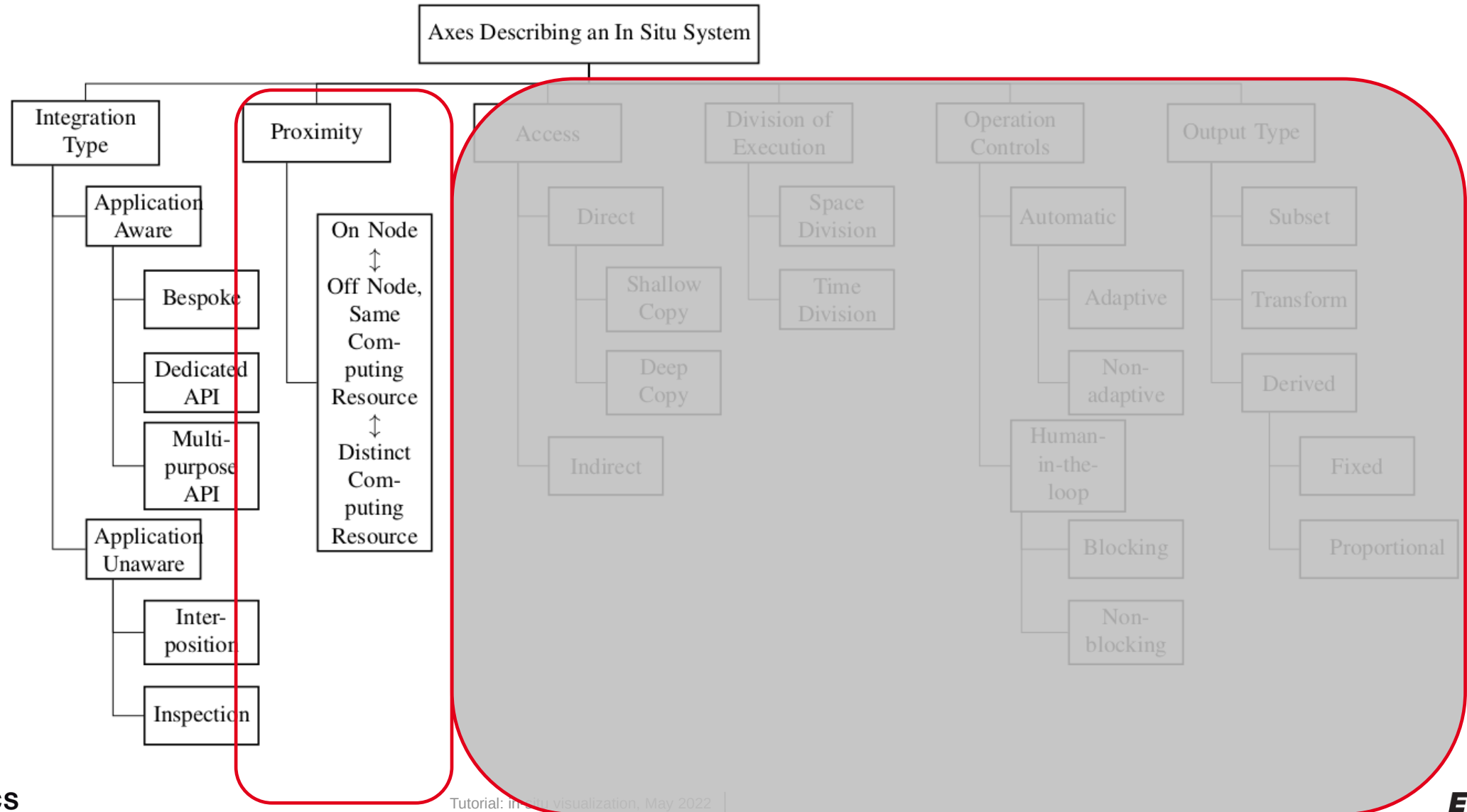
- integration type  
How visualization and analysis code is integrated with the simulation code?
- proximity  
How close is the visualization code from the data?
- Access  
How does the simulation give access to the data?
- division of execution:  
how compute resources are shared between simulation and in situ routines.
- operation controls:  
the mechanism for selecting which operations are executed during run-time
- output type  
which types of operations are performed on the simulation data before it is output.



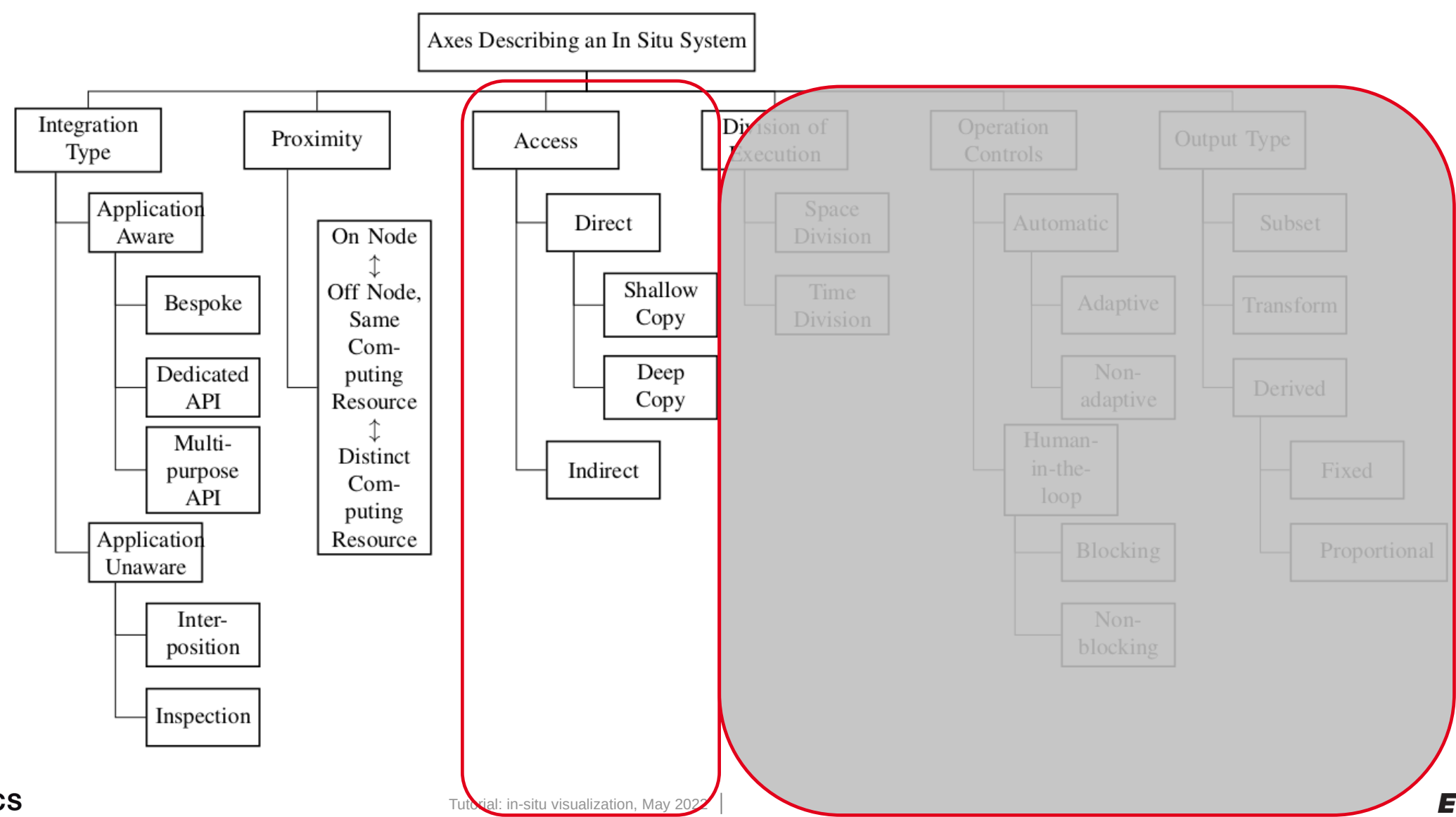
# Integration Type



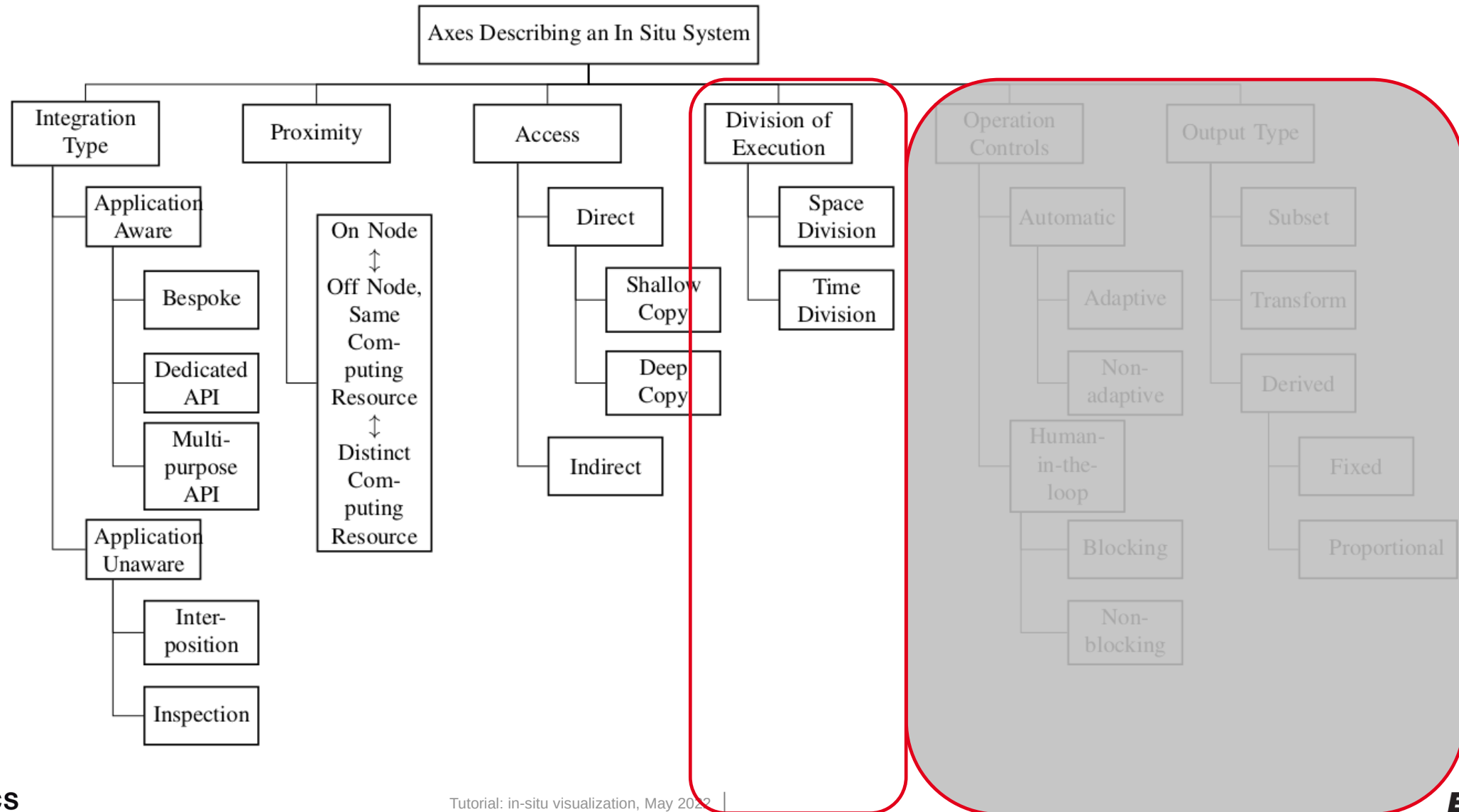
# Proximity



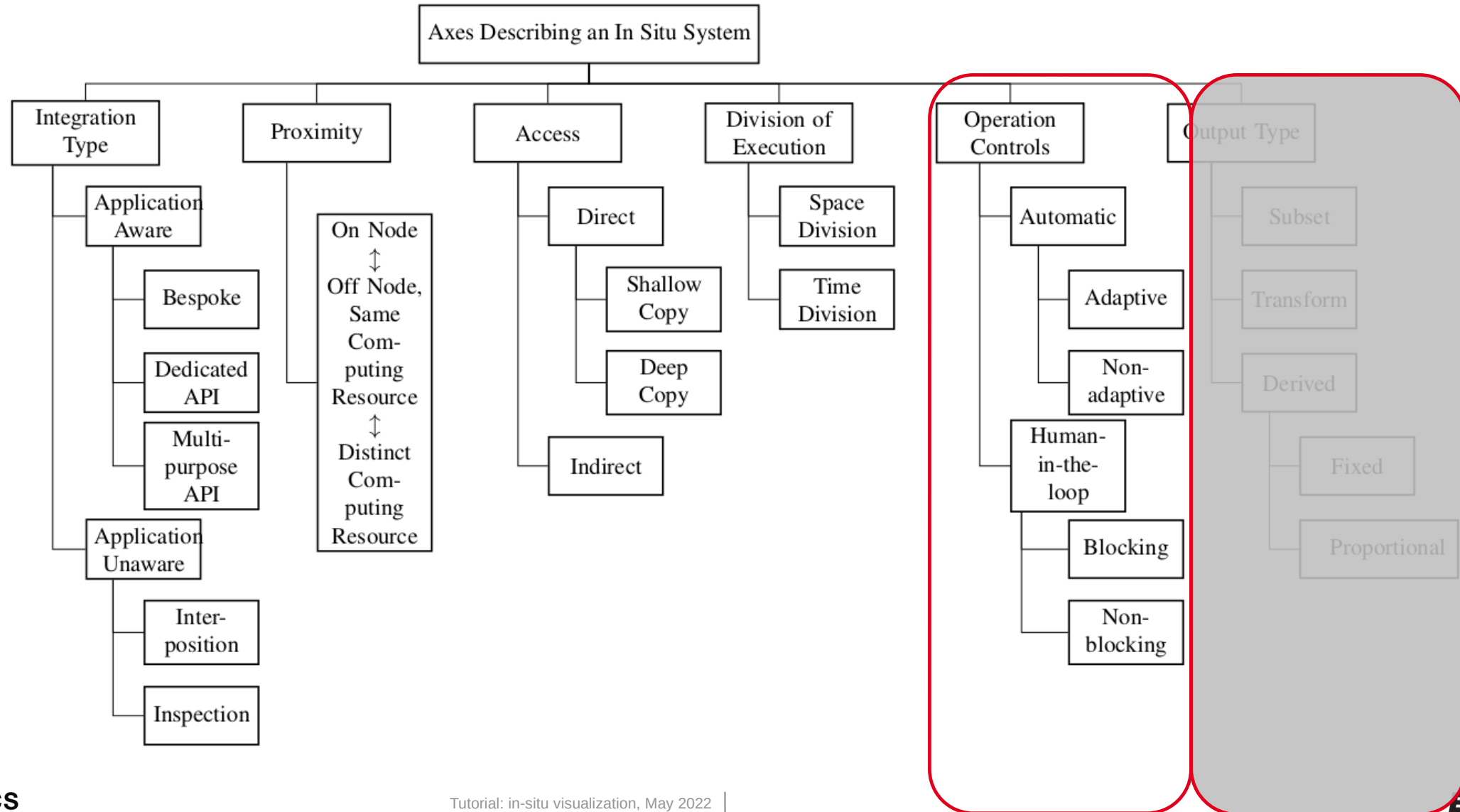
# Access



# Division of Execution

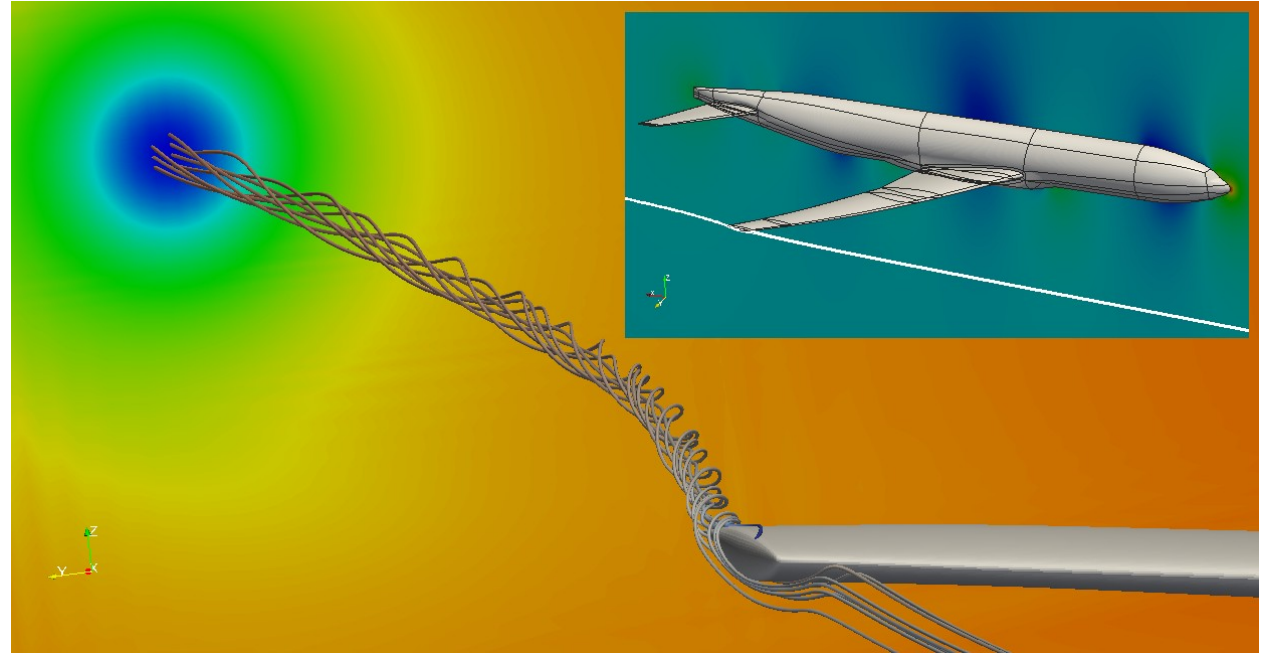


# Operation Controls



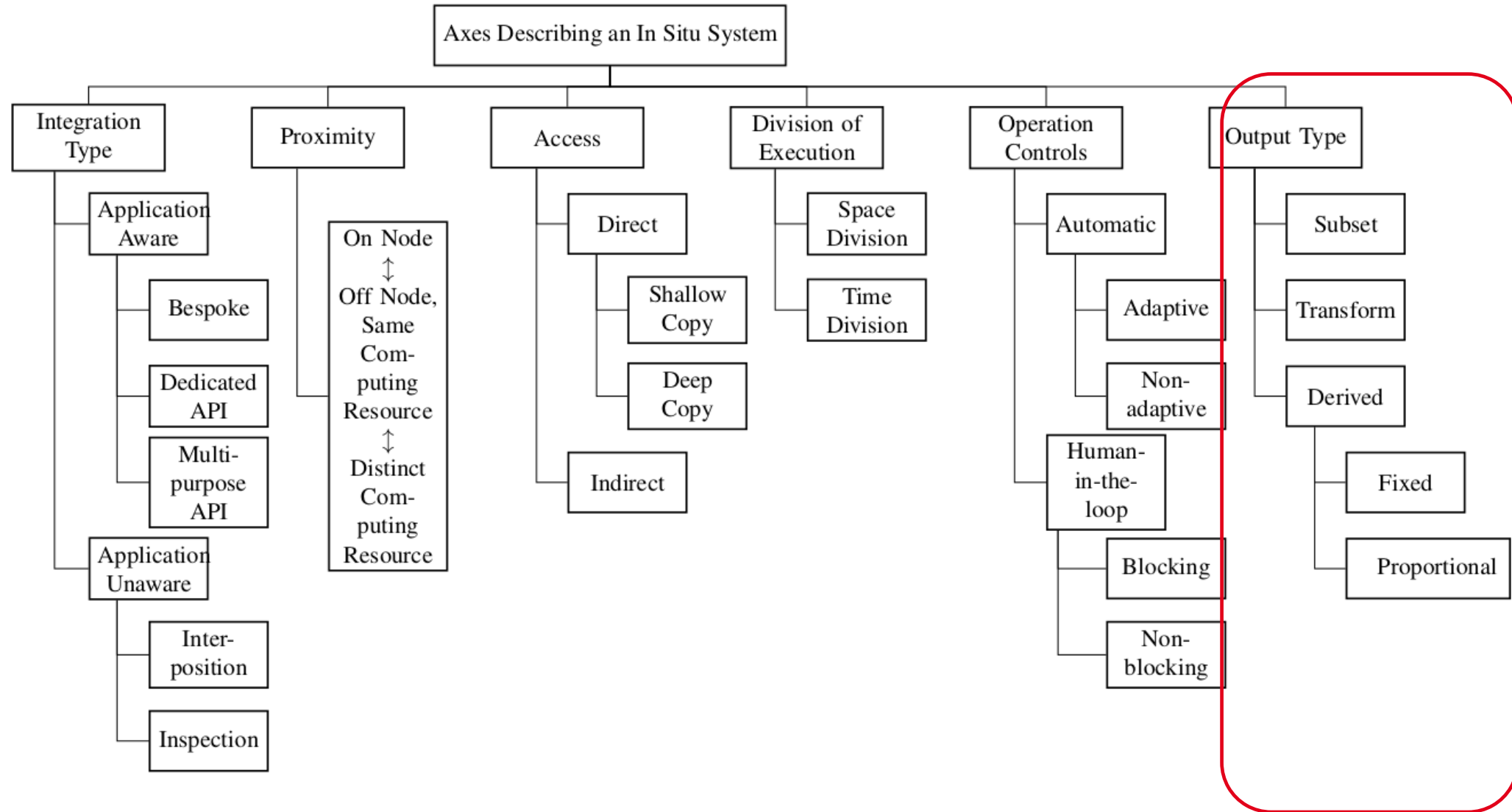
# Feature extraction?

- What's a feature?
- Some small[er] data worth of interest because of domain knowledge



- Without “interactive” data exploration, it can be difficult to know a-priori which features to extract, or how to tune parameters.
- Knowing when to trigger a potentially expensive calculation would also be handy.

# Output types



# Outcome

- In that paper, 15 existing systems were reviewed. Half of them implement a Time Division, with On Node proximity.
  - The simulation advances, then pauses and hands control to the in situ system which completes its operations, hands control back, and so on
  - For example: Ascent, ParaView-Catalyst, Visit/Libsim
- Other examples, ADIOS, SENSEI, which can operate in different modes, sending also their data to distinct vis resources
- ADIOS, for example, can integrate with other workflow or data analytics systems without detailed knowledge of the underlying software and hardware stack
- ADIOS allows users to combine data storage, data staging, data compression, and/or data reduction (ZFP, SZ, BZip2 (compression), FlexPath, Dataspace (data staging), and coupling with ParaView, Visit, Ascent)



# What about my [the solver] internal data model?

- Do the standard visualization applications support all the data structures I use in my code?
  - => Use Data Adaptors
- Are our standard visualization applications ready to handle new alternate data representations?
  - => Use Data Converters
- In recent years, we have seen new ways of thinking about data simulations (run multiphysics code, run ensembles, use ML).
- => new data, perhaps quite different from the traditional "mesh of gridded points"

# Conduit: introduction

---

# Conduit: Simplified Data Exchange for HPC Simulations

- Conduit is an open source project from Lawrence Livermore National Laboratory that provides an intuitive model for describing hierarchical scientific data in C++, C, Fortran, and Python. It is used for data coupling between packages in-core, serialization, and I/O tasks.
- Conduit provides a convention to describe computational simulation meshes. This is called the Mesh Blueprint.
- Illustration of Mesh Blueprint examples
- **Ascent and Catalyst** use **Conduit** for describing data and other parameters which can be communicated between a simulation and the visualization libraries.

# Conduit by examples

Setup from a terminal

```
ssh class5?@daint.cscs.ch
```

```
git clone https://github.com/jfavre/InSitu-Vis-Tutorial2022
```

```
Read and execute contents from "PizDaint_Instructions.txt"
```

# Conduit by examples (C++)

Use our course accounts to get compute nodes on Piz Daint

Username: class5?

Account: crs02

Reservation: insitu

```
srun -C gpu -n 1 -A `id -gn` --time=00:30:00 -p debug --pty bash
```

```
module load daint-gpu Ascent cudatoolkit/21.3_11.2 cray-hdf5-parallel
```

```
cd $EBROOTASCENT/examples/ascent/tutorial/ascent_intro
```

```
cp -r cpp /dev/shm/
```

```
cd /dev/shm/cpp/
```

```
sed -i '34s$/ -L${CRAY_HDF5_PARALLEL_PREFIX}\lib/' Makefile
```

```
make -j8 ASCENT_DIR=$EBROOTASCENT
```

```
./conduit_example1
```

# Conduit by examples (Python)

We will use jupyter lab and our course accounts to get compute nodes on Piz Daint

Username: class5?

Password: ????????

Reservation: insitu

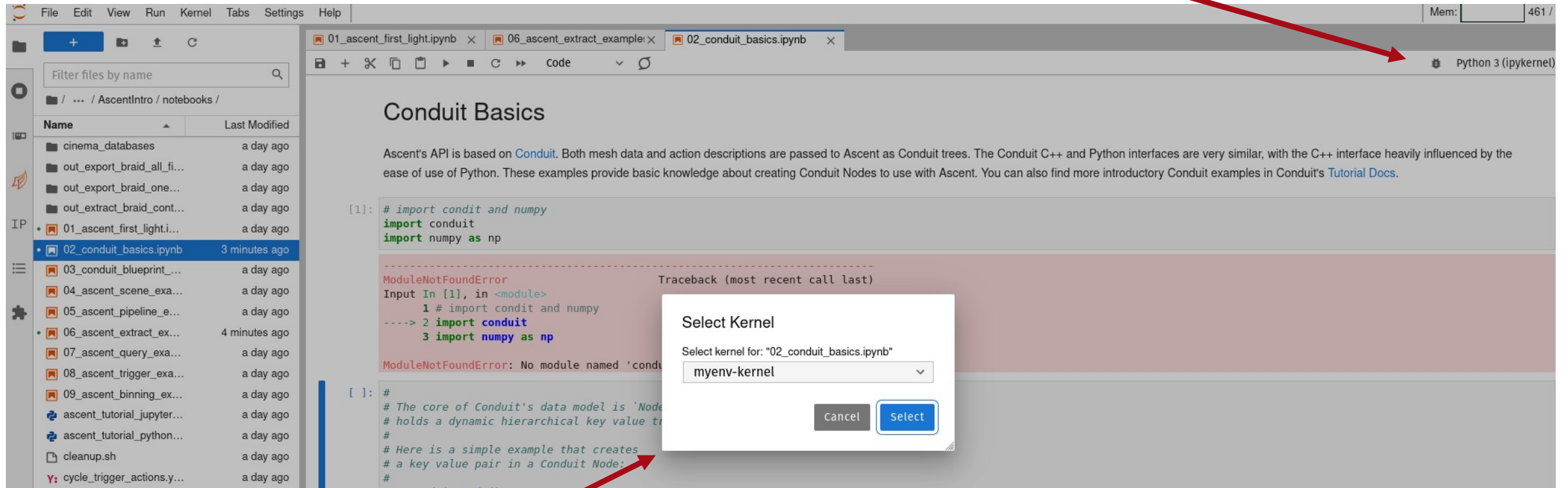
See <https://user.cscs.ch/tools/interactive/jupyterlab/> and get access at

<https://jupyter.cscs.ch>

goto AscentIntro/notebooks

execute 02\_conduit\_basics.ipynb

# import Error?



The screenshot shows a JupyterLab environment. On the left, a file explorer lists notebooks in the directory `/ AscentIntro / notebooks /`. The notebook `02_conduit_basics.ipynb` is selected, showing it was modified 3 minutes ago. The main area displays the notebook's content, which includes a title "Conduit Basics" and a code cell with the following code:

```
[1]: # import conduit and numpy
import conduit
import numpy as np
```

Below the code, a `ModuleNotFoundError` traceback is visible, indicating that the module `conduit` is not found. A "Select Kernel" dialog box is open in the foreground, prompting the user to "Select kernel for: '02\_conduit\_basics.ipynb'". The dropdown menu in the dialog shows `myenvv-kernel` as the selected option. The dialog also includes "Cancel" and "Select" buttons.

Change kernel to *CSCS Python*

# What is available in the CSCS *Python* kernel?

See `$HOME/jupyterlab-cscs.env`

*module unload cudatoolkit*

*module load CMake*

*module load ParaView/5.11.1-CrayGNU-21.09-EGL*

*module load Catalyst/2.0.0-CrayGNU-21.09-rc3*

*export CATALYST\_IMPLEMENTATION\_PATHS=\$EBROOTPARAVIEW/lib64/catalyst*

*module load cudatoolkit/21.3\_11.2*

*module load Ascent/0.9.1-CrayGNU-21.09*

*module load matplotlib/3.5.0-CrayGNU-21.09*

*module load Visit*

*export PMI\_NO\_FORK=1*

*export PMI\_NO\_PREINITIALIZE=1*

*export PMI\_MMAP\_SYNC\_WAIT\_TIME=300*



Coffee break

# Conduit + Ascent

---

# Ascent

Ascent is an easy-to-use flyweight in situ visualization and analysis library for HPC simulations:

- Supports: Making Pictures, Transforming Data, and Capturing Data for use outside of Ascent
- Young effort, yet already includes most common visualization operations
- Provides a simple infrastructure to integrate custom analysis
- Provides C++, C, Python, and Fortran APIs
- Ref

*“The ALPINE in situ infrastructure: Ascending from the ashes of strawman”*, M. Larsen et al., Proc. 3rd Workshop In Situ Infrastructures Enabling Extreme Scale Anal. Vis. Denver, CO, USA, Nov. 12–17, 2017

# Ascent

Ascent slides from the recent SC2021 tutorial

Thanks go to Cyrus Harrison and colleagues.

# Ascent

Ascent is based on several components:

- The Conduit Mesh Blueprint!
- Runtimes providing analysis, rendering and I/O
  - Runtimes will execute a number of *actions*, defined by Conduit Nodes
- Data Adaptors (internal)

# Ascent tutorial pages (Thanks to LLNL)

- [Tutorial link](#)
- [SC2021](#) tutorial
- `docker run -p 8888:8888 -t -i alpinedav/ascent-jupyter`

# Ascent tutorial first example

```
goto AscentIntro/notebooks
```

```
run 01_ascent_first_light.ipynb
```

# Ascent tutorial examples,

## Generating time dependent data

- Run 03\_conduit\_blueprint\_mesh\_examples.ipynb

## Rendering images with Scenes

- Run 04\_ascent\_scene\_examples.ipynb

## Transforming data with Pipelines

- Run 05\_ascent\_pipeline\_examples.ipynb



# Running with different visualization pipelines

In its basic form, we usually have a default scene description into the Ascent bridge code

- At run-time, we can supply different scenes, with additional pipelines
- If present, a file called "ascent\_actions.{json,yaml}" will be read, overriding the actions previously set
- <http://www.yamllint.com/> is your friend
- See example in Examples/LULESH/{ascent\_actions.yaml,trigger\_ascent\_actions.yaml}

# Let's add a scene description

```
"action": "add_scenes",
  "scenes": {
    "s1": {
      "plots": {
        "p1": { "type": "pseudocolor", "field":
"Density" } },
      "renders": {
        "r1": {
          "image_prefix": "DensityImage.%05d",
          "camera": {
            "look_at": [0, 0, 0],
            "position": [-2.17, 1.79, 1.80],
            "up": [0.44, 0.84, -0.30]
          }
        }
      }
    }
  }
```

```
-
action: "add_scenes"
scenes:
  s1:
    plots:
      p1:
        type: "pseudocolor"
        field: "Density"
    renders:
      r1:
        image_prefix: "DensityImage.%05d"
        camera:
          azimuth: 30
          elevation: 11
```

# Let's add a pipeline description

```
"action": "add_pipelines",  
  "pipelines": {  
    "pl1": {  
      "f1": {  
        "type": "threshold",  
        "params": {  
          "field": "Density",  
          "min_value": 1.4,  
          "max_value": 2000  
        }  
      }  
    }  
  }  
},
```

# The scene description is refined with the new pipeline

```
"action": "add_scenes",  
  "scenes": {  
    "s1": {  
      "plots": {  
        "p1": {  
          "type": "pseudocolor",  
          "pipeline": "pl1",  
          "field": "Density"  
        }  
      }  
    },  
  },
```

## Example: Instrument an SPH simulation package with Ascent

- The smooth particle hydrodynamics (SPH) technique is a purely Lagrangian method. SPH discretizes a fluid in a series of interpolation points whose distribution follows the mass density of the fluid.
- PASC, the Swiss Platform for Advanced Scientific Computing initiative, supports the SPH-EXA project developing an SPH library.
- SPH-EXA is a C++17 headers-only code with no external software dependencies. The parallelism is currently expressed via the following models: MPI, OpenMP, CUDA and HIP.

# Instrument the SPH-EXA simulation package with Ascent

- Define a Conduit mesh definition
- Define a Conduit scene definition

About 150 lines of code. Total!

# Using Conduit, a particle set is trivially described [ the coordinates]

```
particle_set = ""  
coordsets:  
  coords:  
    type: "explicit"  
    values:  
      x: [0.0, 10.0, 20.0, 30.0]  
      y: [0.0, 10.0, 20.0, 30.0]  
      z: [0.0, 10.0, 20.0, 30.0]  
""
```

```
conduit::Node mesh;  
mesh["state/cycle"].set_external(&d.iteration);  
mesh["state/time"].set_external(&d.ttot);  
mesh["coordsets/coords/type"] = "explicit";  
mesh["coordsets/coords/values/x"].set_external(&d.x);  
mesh["coordsets/coords/values/y"].set_external(&d.y);  
mesh["coordsets/coords/values/z"].set_external(&d.z);  
// The heavy-data is available via shallow-copy links
```

# Using Conduit, a particle set is trivially described [ the topology]

```
particle_set = ""
topologies:
  mesh:
    type: "unstructured"
    elements:
      shape: "point"
      connectivity: [0, 1, 2, 3]
      coordset: "coords"
""
```

```
mesh["topologies/mesh/type"].set("unstructured");
mesh["topologies/mesh/elements/shape"].set("point");
mesh["topologies/mesh/coordset"].set("coords");

std::vector<int> conn(N); // N is # of particles
std::iota(conn.begin(), conn.end(), 0);
mesh["topologies/mesh/elements/connectivity"].set_external(conn);
```



# Using Conduit, a particle set is trivially described [ the solution fields]

```
particle_set = ""
fields:
  rho:
    association: "vertex"
    values: [-1, -2, -3, -4]
    topology: "mesh"
    volume_dependent: "false"
    units: "g/cc"
""
```

```
auto fields = mesh["fields"];
// Density scalar field
fields["rho/association"].set("vertex");
fields["rho/topology"].set("mesh");
fields["rho/volume_dependent"].set("false");
// Conduit supports shallow copy
fields["rho/values"].set_external(&d.rho);
```

# Use one of conduit's relay protocol to save data to disk?

c++: [extract node from SPH-EXA](#)

---

In Python

```
import conduit.relay as relay
if relay.io.about()["protocols/hdf5"] == "enabled":
    relay.io.save(mesh, "/dev/shm/foo.hdf5")
```

```
h5ls -r /dev/shm/foo.hdf5
/
/coordsets
/coordsets/coords
/coordsets/coords/type
/coordsets/coords/values
/coordsets/coords/values/x
...
/fields
/fields/rho
/fields/rho/values
....
/topologies
/topologies/mesh
/topologies/mesh/coordset
/topologies/mesh/elements
/topologies/mesh/elements/connectivity
```

# Supplementary demonstration material

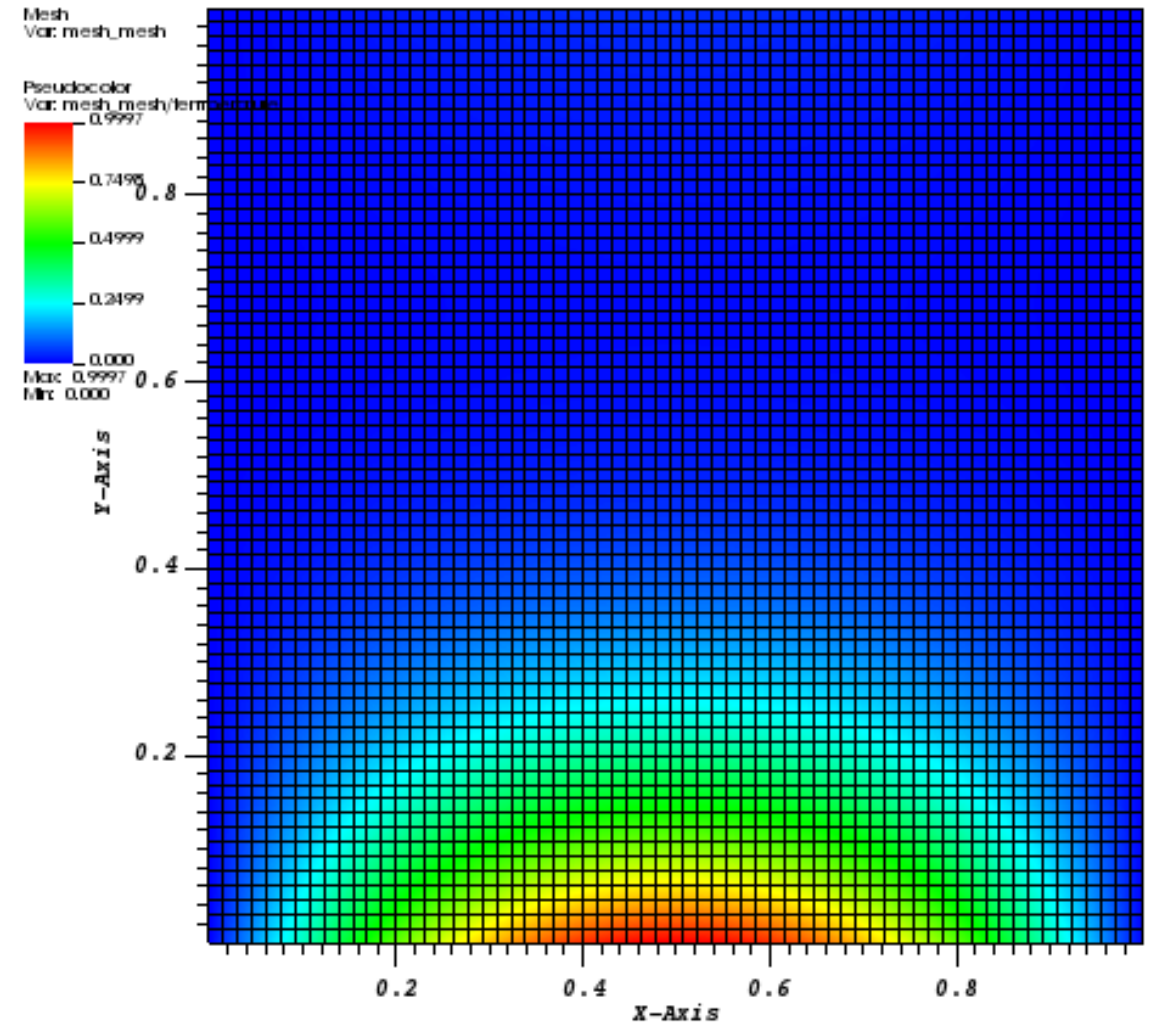
---

# Instrument the Jacobi-Python example with Ascent

Define a Conduit mesh definition

Define a Conduit scene definition

DB: mesh.cycle\_000500.root  
Cycle: 500



# Instrument the Jacobi-Python example with Ascent

- For demonstration purposes, we can choose 4 different grid types:
- "uniform", "rectilinear", "structured", "unstructured"

`Simulation_With_Ascent(meshtype="uniform")`

- Study the files *rectilinear.yaml* *structured.yaml*  
*uniform.yaml* *unstructured.yaml*

```
coordsets:  
  coords:  
    type: "uniform"  
    dims:  
      i: 6  
      j: 6  
    origin:  
      x: 0.0  
      y: 0.0  
    spacing:  
      dx: 0.2  
      dy: 0.2  
  topologies:  
    mesh:  
      type: "uniform"  
      coordset: "coords"  
  fields:  
    temperature:  
      association: "vertex"  
      topology: "mesh"  
      Values: [0.0, 0.587785252292473,  
0.951056516295154, ...]
```

## Examples/JacobiPython/jacobi\_insitu\_Ascent.ipynb

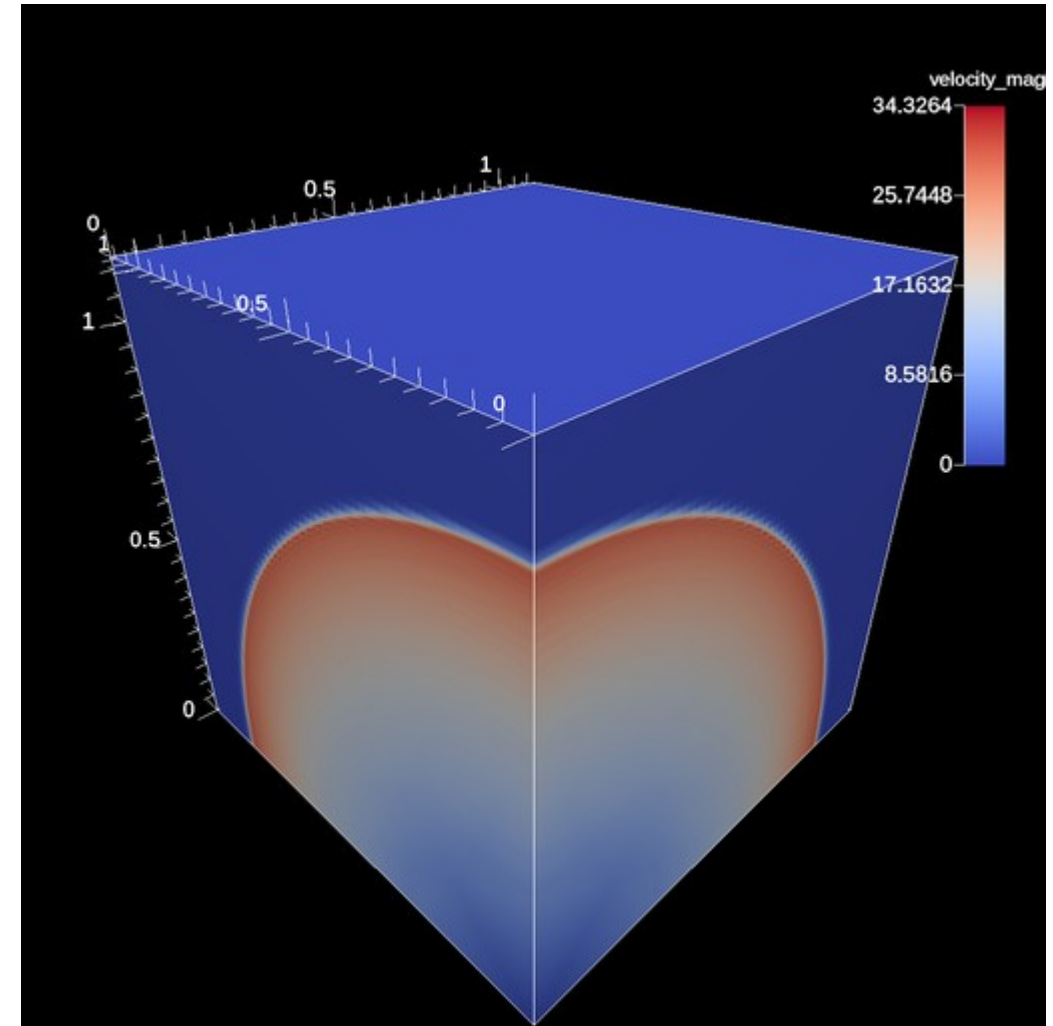
- The notebook demonstrates one additional feature:
- An Ascent „extract“ action used to save the final grid data and the VisIt app reading back the Blueprint HDF5 mesh

```
action = conduit.Node()
add_extr = action.append()
add_extr["action"] = "add_extracts"
extracts = add_extr["extracts"]
extracts["e1/type"]="relay"
extracts["e1/params/path"] = savedir + "mesh";

extracts["e1/params/protocol"] = "blueprint/mesh/hdf5";
self.a.execute(action)
```

# Instrument the LULESH example with Ascent

- Define a Conduit mesh definition
- Define a Conduit scene definition



# Execute the LULESH code with Ascent

- Change directory to Examples/LULESH
- See ASCENT\_HowtoCompile\_and\_Run\_PizDaint.txt
- Run the standard job (see Ascent/Run\_LULESH+Ascent.ipynb)
- Exercise
- Instead of plotting the velocity\_magnitude, compute first isocontours(velocity\_magnitude) for a fixed range of values, and plot these
- Hint: Run 05\_ascent\_pipeline\_examples.ipynb
- Hint: <https://ascent.readthedocs.io/en/latest/Actions/Examples.html#an-example-of-the-contour-filter-with-a-multiple-iso-values>



# Lunch Break

---

**Before jumping to ParaView Catalyst, let's first introduce ParaView**

---

# ParaView in one slide

- ParaView [Introduction](#)
- [Paraview Architecture](#)
- [Parallel Rendering](#)
- Python & Batch: [ParaView & Python](#)
- Python & Batch: [pvpython and pvbatch](#)

# Conduit + ParaView Catalyst

---

# Catalyst : a VTK-based API

vs. a Conduit-based API

## Original API (ParaView before v5.9)

- Writing a data adaptor requires an intimate understanding of VTK. VTK Mesh types, Data types.
- Knowledge of how to efficiently transform simulation data structures to VTK is a must.
- Not for the faint-at-heart
- Not for a simulation code developer, without a hobby for Visualization software development

# Catalyst : a VTK-based API

# vs. a Conduit-based API

## Original API (ParaView before v5.9)

- Writing a data adaptor requires an intimate understanding of VTK. VTK Mesh types, Data types.
- Knowledge of how to efficiently transform simulation data structures to VTK is a must.
- Not for the faint-at-heart
- Not for a simulation code developer, without a hobby for Visualization software development

## (with v5.9 and onwards)

- Writing an adaptor requires a discreet knowledge of Conduit, which I acquired in one afternoon.
- Catalyst is a lightweight API and a simulation can be linked against its stub implementation, without requiring a ParaView SDK
- A ParaView 5.9+ build provides a Catalyst API implementation, which is now referred to as *ParaView-Catalyst*
- [Catalyst in ParaView 5.9: Blog article](#)

# ParaView Catalyst

- ParaView-Catalyst is an implementation of the Catalyst *in situ* API that uses ParaView for data processing and rendering.
- ParaView-Catalyst supports a subset of the [Mesh Blueprint](#). Simulations that can use the Mesh Blueprint to describe their data can directly use ParaView's Catalyst implementation for in situ analysis and visualization.
- [ParaView-Catalyst](#)
- [ParaView-Catalyst Blueprint](#)

# ParaView Catalyst practical how-to

- Instrumented our simulation code? Yes!
- Compiled with ParaView Catalyst? Yes!
- How do we run and specify visualization tasks????



# The ParaView Extractors

- Extractors are items in the visualization pipeline that can save data or images at a user-chosen temporal frequency.
  - Data Extractors
  - Image Extractors
- Using Extractors, no custom code per iteration is really necessary in the majority of cases. One can simply use Extractors to save out images from views or data extracts from filters and other data producers. (*more on that later*)

# The ParaView Catalyst Python scripts

- Python scripts, are written by the ParaView application, given a representative template input file, and a set of visualization filters interactively tuned by the user in an *offline* fashion [not connected to a running solver]
- We create a visualization pipeline with the numerous visualization filters and rendering options available in ParaView, and we add “Extractors”
- ParaView provides hybrid parallelism out-of-the-box
  - MPI-enabled
  - TBB multi-threading
  - CUDA-enabled filters
- Some [or most of the] ParaView visualization code will be tightly integrated with the solver memory and execution space.

# ParaView Catalyst Python scripts

- Can be generated from an interactive ParaView session, and later fine-tuned
- Are completely interchangeable between the batch-mode ParaView execution (reading data from disk), and the in-situ execution

## Catalyst scripts (batch-mode

vs.

## in-situ mode)

```
grid = OpenDataFile(registrationName='grid',  
filename=['/LULESH/datasets/data_000009.vtpd'])
```

```
renderView1 = GetRenderView()
```

```
rep = Show()
```

```
ColorBy(rep, ['POINTS', 'velocity'])
```

```
Render()
```

```
# execute in batch-mode with data read from disk
```

```
from paraview.simple import
```

```
    SaveExtractsUsingCatalystOptions
```

```
SaveExtractsUsingCatalystOptions(options)
```

```
grid = TrivialProducer(registrationName='grid')
```

```
renderView1 = GetRenderView()
```

```
rep = Show()
```

```
ColorBy(rep, ['POINTS', 'velocity'])
```

```
v = CreateExtractor('VTPD', grid)
```

```
v.Trigger = 'TimeStep'
```

```
v.Trigger.Frequency = 30
```

```
v.Writer.FileName = 'data_{timestep:06d}.vtpd'
```

# ParaView-Catalyst Blueprint

The Protocol is rather simple:

- Defines the options accepted by **catalyst\_initialize()**; these include things like ParaView Python scripts to load, directories to save data
- Defines the protocol for **catalyst\_execute()** and includes information about Catalyst channels i.e. ports on which data is made available to in situ processing as well as the actual data from the simulation
- Defines the protocol for **catalyst\_finalize()**

# ParaView-Catalyst Blueprint

Similar to the Conduit Mesh Blueprint.

```
node["catalyst/scripts/script/filename"] =
```

```
node["catalyst/state/cycle"] =
```

```
node["catalyst/state/time"] =
```

```
node["catalyst/channels/grid/type"] = "mesh"
```

```
node["catalyst/channels/grid/data"] =
```

## Code instrumentation -

The Catalyst glue code for the SPH-EXA solver is 144 lines of code

Enabling in-situ visualization can be optionally compiled

## before

```
int main(int argc, char** argv)
{
    MPI_Init_and_Code_Init();

    for (d.iteration = 0; d.iteration <= maxStep; d.iteration++)
    {
        Solve_For_Each_Timestep();

    }

    return exitSuccess();
}
```

## Code instrumentation -

- The Catalyst glue code for the SPH-EXA solver is 144 lines of code
- The execution driver is instrumented with 4 lines of code
- Total: 148 lines of code

## after

```
#include "CatalystAdaptor.h"
int main(int argc, char** argv)
{
    MPI_Init_and_Code_Init();
    CatalystAdaptor::Initialize(argc, argv);
    for (d.iteration = 0; d.iteration <= maxStep; d.iteration++)
    {
        Solve_For_Each_Timestep();
        CatalystAdaptor::Execute(d, domain.startIndex());
    }
    CatalystAdaptor::Finalize();
    return exitSuccess();
}
```

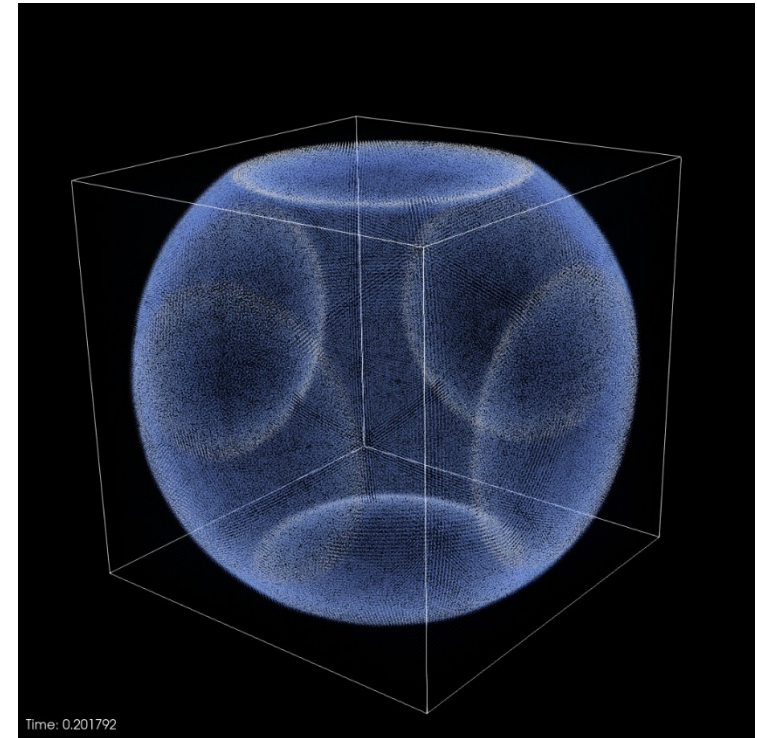


# What about “Operations Control”?

## Adaptive Control

- Choose different pathes of execution based on queries/triggers
- The `catalyst_execute(ConduitNode)` script can be customized

```
def catalyst_execute(node):  
    threshold = 1.4  
    if reader.PointData["Density"].GetRange()[1] > threshold:  
        print("density at timestep", node.timestep)  
        # Extract particles where Density > threshold  
    else:  
        # use ALL particles
```



# What about “Operations Control”?

## Human in the loop

ParaView has a very efficient client-server mode for post-hoc visualization

- Uses a powerful, remote server to do the heavy work:
  - Parallel I/O
  - Parallel filtering
  - Parallel Image Composition
- Has a very efficient, threshold-based, **Image Delivery** engine
- The results of the different visualization operations stay on the server (Mbytes, Gbytes, Tbytes (choose your flavour))
- Only the image gets sent over the network to the client (1024x1024 pixels, RGB).

# ParaView-Catalyst uses the same engine

- Open a port on a reverse tunnel to my desktop
  - `ssh -R 22222:localhost:22222 daint103.cscs.ch`
  - Tell my desktop client to send a request for connection on that port
  - my code instrumentation listens to incoming calls with four additional lines of Python code:

```
from paraview import catalyst  
options = catalyst.Options()  
options.EnableCatalystLive = 1  
options.CatalystLiveURL = 'daint103:22222'
```

## Demonstration of Catalyst Live

# Other examples: LULESH instrumented for ParaView Catalyst

## ASC Proxy Apps

Instrumented with the VTX-based Catalyst

Instrumented with the Conduit-based Catalyst

Thanks to Utkarsh Ayachit (Kitware)

# Execute the LULESH code with Catalyst

- Change directory to Examples/LULESH
- See ASCENT\_HowtoCompile\_and\_Run\_PizDaint.txt
- Run the standard job (see Ascent/Run\_LULESH+Catalyst.ipynb)
- Exercise

# ParaView pipeline

vs.

# Ascent pipeline

```
renderView1 = CreateView('RenderView')

selection=SelectPoints()
selection.QueryString="Density >= 1.4"

extractSelection = ExtractSelection()
thresholdDisplay = Show(extractSelection)
ColorBy(thresholdDisplay, ['POINTS', 'Density'])

pNG1 = CreateExtractor('PNG', renderView1)
pNG1.Trigger = 'TimeStep'
pNG1.Writer.FileName =
'threshold_{timestep:06d}{camera}.png'
pNG1.Trigger.Frequency = 100
```

```
"action": "add_pipelines",
  "pipelines": {
    "pl1": {
      "f1": {
        "type": "threshold",
        [...]
      }
    }
  },
  "action": "add_scenes",
  "scenes": {
    "s1": {
      "plots": {
        "p1": {
          "type": "pseudocolor",
          "pipeline": "pl1",
          [...]
        }
      }
    }
  },
  "renders": {
    "r1": {
      "image_prefix": "ThresholdImage.%05d",
```

## Third Method. Use Ascent and ParaView Python scripts

---

# Ascent, passing data to a ParaView script

- [Ascent Documentation](#)
- The instrumented simulation sends a tree structure (json like) that describes the simulation data using the Conduit Blueprint Mesh specification.
- A ParaView plugin constructs one of the following datasets: vtkImageData, vtkRectilinearGrid, vtkStructuredGrid or vtkUnstructuredGrid
- This data is converted to a VTK format using shallow copies for data arrays



# Ascent, passing data to a ParaView script

- Ascent's “action” node is:

```
"action": "add_extracts",  
  "extracts":  
  {  
    "e1":  
    {  
      "type": "python",  
      "params":  
      {  
        "file": "paraview-vis.py"  
      }  
    }  
  }  
}
```

- The ParaView script describes a “standard” ParaView pipeline and can save images and any other derived data to disk

# Execute the LULESH code with Ascent+ParaView plugin

- Change directory to Examples/LULESH/Ascent/ParaViewBridge
- 
- Run the notebook (Run\_LULESH+Ascent+ParaView.ipynb)

# Summary

- Adopting the two-Conduit-based environments presented is rather simple, very intuitive. The bridges to ParaView or Ascent are now hiding all the complexity from the code developers
- Both environments have proven tracks of high-scale deployments
- What's next:
  - evaluate performance, memory consumption,
  - visualization pipeline creation and tuning, and re-usability
- For the first time in several decades of developing data formats plugins (converters, adaptors, ..), I don't have to worry about this anymore, and can concentrate on what I love to do best: !! Visualization !!
- Already 10 years ago, while presenting libSim, people would say

*“we’re **very** tight on memory”*

- Adding an *in-transit* layer is the next venue to explore.

## In-transit visualization. Recipe for a live demo. Part 1

- Use [ADIOS2](#) and the new [plugin interface](#) which brings the ability for users to load their own engines and operators through the plugin interface.
- The short summary about what ADIOS is and isn't is [here](#)
- An application compiled with ADIOS can [change run-time parameters](#) at start-time without the need to recompile.
- We can use a specific [Engine](#) (SST) to stream data.
- The Sustainable Staging Transport (SST) is an engine that allows direct connection of data producers and consumers via the ADIOS2 write/read APIs.
- A classic streaming data architecture where the data passed to ADIOS on the write side is made directly available to a reader app.

## In-transit visualization. Recipe for a live demo. Part 2

- Fides enables complex scientific workflows to seamlessly integrate simulation and visualization. This is done by providing a data model in JSON that describes the mesh and fields in the data to be read.
- Using this data model, Fides maps ADIOS2 data arrays (from files or streams) to VTK-m datasets, enabling visualization of the data using shared- and distributed-memory parallel algorithms.
- Fides is available in ParaView

# In-transit visualization. Recipe for a live demo. Part 3

- Demonstrating on my laptop. [Docs](#)
- A parallel simulation, writing ADIOS2 data to disk
- `mpiexec -n 4 ./gray-scott settings-staging.json`

```
{  
  "L": 64,  
  "Du": 0.2,  
  "Dv": 0.1,  
  "F": 0.01,  
  "k": 0.05,  
  "dt": 2.0,  
  "plotgap": 10,  
  "Steps": 1000,  
  "output": "gs.bp",  
  "checkpoint_output": "gs_ckpt.bp",  
  "adios_config": "adios2-fides.xml",  
  "mesh_type": "image"  
}
```

```
<?xml version="1.0"?>  
<adios-config>  
  <io name="SimulationOutput">  
    <engine type="BP4">  
      <parameter key="verbose" value="5"/>  
      <parameter key="RendezvousReaderCount" value="1"/>  
      <parameter key="QueueLimit" value="5"/>  
      <parameter key="QueueFullPolicy" value="Block"/>  
    </engine>  
  </io>  
<?xml version="1.0"?>
```

# In-transit visualization. Part 3 output to disk

- `/local/apps/ADIOS2-v2.9.0Build/bin/bpls -a gs.bp`
- 
- `double Du attr`
- `double Dv attr`
- `double F attr`
- `string Fides_Data_Model attr`
- `string Fides_Dimension_Variable attr`
- `double Fides_Origin attr`
- `double Fides_Spacing attr`
- `string Fides_Variable_Associations attr`
- `string Fides_Variable_List attr`
- `double U 100*{64, 64, 64}`
- `double V 100*{64, 64, 64}`

ParaView can now read the data from disk.

This is the good-old **post-hoc** visualization paradigm!

# In-transit visualization. Recipe for a live demo. Part 3

- Streaming ADIOS2 data between one producer and one consumer
- `mpiexec -n 4 ./gray-scott settings-staging.json`
- `mpiexec -n 2 ./fides-sst-reader`

```
{  
  "L": 64,  
  "Du": 0.2,  
  "Dv": 0.1,  
  "F": 0.01,  
  "k": 0.05,  
  "dt": 2.0,  
  "plotgap": 10,  
  "Steps": 1000,  
  "output": "gs.bp",  
  "checkpoint_output": "gs_ckpt.bp",  
  "adios_config": "adios2-fides.xml",  
  "mesh_type": "image"  
}
```

```
<?xml version="1.0"?>  
<adios-config>  
  <io name="SimulationOutput">  
    <engine type="SST">  
      <parameter key="verbose" value="5"/>  
      <parameter key="RendezvousReaderCount" value="1"/>  
      <parameter key="QueueLimit" value="5"/>  
      <parameter key="QueueFullPolicy" value="Block"/>  
    </engine>  
  </io>  
<?xml version="1.0"?>
```



# In-transit visualization. Recipe for a live demo. Part 4

- Switching Engine to a ADIOS2 “ParaView” plugin
- `mpiexec -n 4 ./gray-scott settings-inline.json`

```
{  
  "L": 64,  
  "Du": 0.2,  
  "Dv": 0.1,  
  "F": 0.01,  
  "k": 0.05,  
  "dt": 2.0,  
  "plotgap": 10,  
  "steps": 1000,  
  "output": "gs.bp",  
  "checkpoint_output": "gs_ckpt.bp",  
  "adios_config": "adios2-inline-plugin.xml",  
  "mesh_type": "image"  
}
```

```
<?xml version="1.0"?>  
<adios-config>  
  <io name="SimulationOutput">  
    <engine type="plugin">  
      <!-- general plugin engine parameters -->  
      <parameter key="PluginName" value="fides"/>  
      <parameter key="PluginLibrary"  
value="ParaViewADIOSInSituEngine"/>  
      <!-- ParaViewFides engine parameters -->  
      <parameter key="DataModel" value="gs-catalyst-fides.json"/>  
      <parameter key="Script" value="gs-catalyst-fides.py"/>  
    </engine>  
  </io>  
<?xml version="1.0"?>
```

# Resources

- <https://github.com/jfavre/InSitu-Vis-Tutorial2022>
- [My slides](#)
- [https://dav.lbl.gov/events/SC21\\_SENSEI\\_Ascent\\_Tutorial/](https://dav.lbl.gov/events/SC21_SENSEI_Ascent_Tutorial/)

---

**Thank you for your attention.**